特別研究報告

題目

Docker イメージのレイヤ構造に着目した Dockerfile の自動テスト生成手法の提案

指導教員

楠本 真二 教授

報告者

後藤 有希

令和 7 年 2 月 6 日

大阪大学基礎工学部情報科学科

令和 6 年度 特別研究報告

Docker イメージのレイヤ構造に着目した Dockerfile の自動テスト生成手法の提案

後藤 有希

内容梗概

軽量なコンテナ型仮想化用のプラットフォームとして Docker が注目されている. Docker では Docker イメージの構築手順を Dockerfile と呼ばれるテキストファイルに記述する．Dockerfile は専用の命令が記述された一種のソースコードであり，一般的なプログラミング言語と同様にテストを用いてその振る舞いを検証すべきである．Java などのソースコードに対しては，探索に基づいた全自動でのテスト生成手法が提案されている．しかし従来のテスト生成手法を Dockerfile に適用することはできない．Dockerfile には分岐が存在せず，自動テスト生成の目的関数となる網羅率が意味をなさないためである．本研究では，処理手順ではなく処理結果に基づいて Dockerfile のテストを自動生成する．提案手法では Dockerfile 命令と Docker イメージのレイヤの解析に基づいてテスト対象を決定し，テストを生成する．評価実験の結果，開発者が作成したテストの最大 8 割以上をカバー可能であることが確認できた．

主な用語

Docker，Docker イメージ，Dockerfile，ソフトウェアテスト，自動テスト生成

目次

図目次

表目次

1　はじめに

軽量なコンテナ型仮想化用のプラットフォームとして Docker が注目されている．Docker ではコンテナ型仮想化によりアプリケーションを開発環境や実行環境から切り離す．これにより再現性と可搬性の向上，更には迅速なデプロイを実現している [1]．このような利点から，Docker はソフトウェア開発で利用されている [2] ほか，学術分野において計算の再現性を確保するための手段としても注目されている [3]．Docker では Docker イメージの構築手順を Dockerfile と呼ばれるテキストファイルに記述する．この Dockerfile をビルドすることで Docker イメージが得られる．更にこの Docker イメージを実行することでコンテナと呼ばれる仮想環境が起動する．

Dockerfile は専用の命令が記述された一種のソースコードであり，一般的なプログラミング言語と同様にテストを用いてその振る舞いを検証すべきである．Dockerfile ではパッケージマネージャによるインストールや，`wget` コマンドによるダウンロードなど通信を必要とするコマンドが多用される．そのため Dockerfile 内部の記述内容ではなく外的な要因によってデグレードが発生しやすい [4]．Henkel らは GitHub 上にある Dockerfile の 26% がビルドに失敗することを確認しており，多くは依存関係をはじめとする外部環境の変化が原因であった [4]．また Dockerfile に対してテストを用意することで，リファクタリング時の振る舞い保持の検証にも利用可能である．

Java などのプログラミング言語では，開発者によるテスト作成支援を目的とした自動テスト生成手法が多数提案されている [5][6]．自動生成されたテストはデグレード対策やリファクタリングといった場面で活用される．これらの手法ではメソッドの引数に入力値として様々な値を与え，実行経路に着目して網羅率を最大化するように探索する．例えば Java に対する自動テスト生成ツールとしては，探索ベースのアルゴリズムを活用した EvoSuite[7] が広く知られている．EvoSuite は高い網羅率を持つテストの生成が可能であり，大規模な実証実験においてもクラスあたり平均 71% の分岐網羅率を達成できることが示されている [8]．Dockerfile に対してもこのように自動テスト生成による開発者支援が可能だと考える．

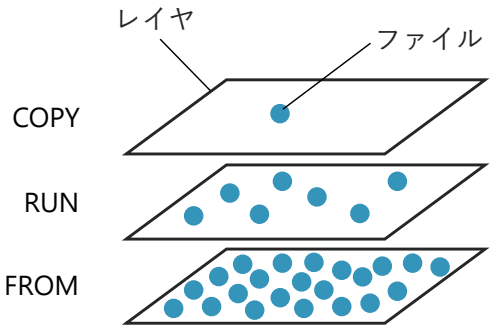
しかし，既存の探索ベースの自動テスト生成手法は Dockerfile には適応できない．これは Dockerfile には分岐が存在せず実行経路が 1 つしかないため，既存の自動テスト生成手法で目的関数としていた網羅率が意味をなさないことが原因である．よって既存手法の核となる探索が応用できず，Dockerfile の自動テスト生成手法には異なるアイデアが必要となる．

本研究では，Dockerfile の自動テスト生成手法を提案する．提案手法の鍵となるアイデアは処理手順ではなく処理結果に基づいてテストを生成する点である．Dockerfile から構築された Docker イメージを Dockerfile の作用の集合だと考え，Dockerfile 命令と Docker イメージのレイヤの解析に基づいてテスト対象を決定し，テストを生成する．評価実験の結果，提案手法により開発者が作成したテストの最

大 8 割以上をカバー可能であることが確認できた.

```
# ベースイメージの指定
FROM debian:latest
# Pythonのインストール
RUN apt-get update && \
    apt-get install -y \
    python3
# main.pyのコピー
COPY main.py .
# 起動時にスクリプト実行
CMD ["python3", "main.py"]
```

(a) Dockerfile の例



(b) 構築される Docker イメージ

図 1: ビルド実行例

2 準備

2.1 Dockerfile とレイヤ

Docker では Docker イメージの構築手順を Dockerfile と呼ばれるファイルに記述する．Dockerfile の記述例を 図 1(a) に示す．図 1(a) の Dockerfile には Python 実行環境の構築，及び特定のスクリプトを実行する手順が記述されている．最初の FROM 命令で最も基礎となるイメージ（ベースイメージ）を Debian に指定し，次の RUN 命令内でシェルコマンドを使用して Python をインストールする．そして COPY 命令で必要なファイルを追加し，最後に CMD 命令でコンテナ起動時に実行するシェルコマンドを設定している．RUN 命令の中では 図 1(a) のように “&&” を用いて複数のシェルコマンドを記述できる．この Dockerfile をビルドすることで 図 1(b) に示すような Docker イメージが得られる．Docker イメージは構築した環境の一種のスナップショットであり，この Docker イメージを実行することでコンテナと呼ばれる仮想環境が起動する．

Dockerfile から構築される Docker イメージの実態はレイヤ構造のファイルシステムであり，各レイヤは Dockerfile 命令と対応している．Dockerfile のビルドを開始すると Dockerfile 命令は上から順に実行され，各命令で追加されるファイルなどの変更差分がレイヤとなる．図 1(a) の Dockerfile の場合，まず FROM 命令が実行されると，図 1(b) のように FROM 命令に対応したレイヤにベースイメージに必要なファイルが大量に追加される．次に RUN 命令の実行により Python をインストールすると，RUN 命令に対応したレイヤが上に重なり，そのレイヤの中に `python3` コマンドに加えて関連ファイルや `apt-get` のキャッシュが追加される．そして COPY 命令を実行すると対応したレイヤが更に上に重なり，レイヤには `main.py` が追加される．最後の CMD 命令ではコンテナのメタ情報の設定のみ行い，

```yaml
schemaVersion: '2.0.0'
commandTests:
  - name: 'check python3'
    command: 'python3'
    args: ['--version']
    expectedOutput: ['3\.11\..*']
fileExistenceTests:
  - name: 'check main.py'
    path: 'main.py'
    shouldExist: true
metadataTest:
  cmd: ['python3', 'main.py']
```

図 2: Container Structure Test を用いた 図 1(a) の Dockerfile に対するテストの一例

ファイルの変更差分はないためレイヤは生成されない．これらのレイヤを透過的に重ね合わせることにより，1 つのファイルシステムを構築している．

2.2 Container Structure Test

Container Structure Test (CST) [*1] は Dockerfile に対するテストフレームワークである．CST を利用することで，起動したコンテナに対するシェルコマンドの実行結果やファイルの存在などを検証できる．CST では以下 6 種類の検証方法が利用できる．

- **commandTests**：シェルコマンドの実行結果を検証
- **fileExistenceTests**：ファイルの存在を検証
- **fileContentTests**：ファイルの内容を検証
- **metadataTest**：コンテナのメタ情報を検証
- **licenseTests**：ライセンスファイルを検証
- **globalEnvVars**：環境変数を検証

図 1(a) の Dockerfile に対する CST を用いたテストの一例を 図 2 に示す．1 つ目の commandTests は RUN 命令で Python を正しくインストールできたか確認するために，`python3` コマンドにオプション`--version` を指定して実行し，バージョンが 3.11 であることを検証している．2 つ目の fileExistenceTests は COPY 命令で追加した `main.py` がコンテナに存在することを検証している．3 つ目の metadataTest は CMD 命令で指定したシェルコマンドが正しく設定されているか検証してい

[*1] `https://github.com/GoogleContainerTools/container-structure-test`

る．本研究ではこの CST 形式で Dockerfile のテストを自動生成する．

2.3 既存の自動テスト生成手法とその課題

自動テスト生成とは，ソースコードから単体テストをボトムアップに生成する技術である．これまで様々なプログラミング言語に対する自動テスト生成手法が提案されており，Java では EvoSuite[7]，Python では Pynguin[9] などが有名である．既存の自動テスト生成手法ではメソッドの引数に与える入力値に応じた実行経路に着目している．様々な入力値を与えることで実行経路の網羅率を最大化するように探索する．しかし，Dockerfile は Java などのプログラミング言語と性質が異なり，このような従来の自動テスト生成手法を適応できない．大きな要因として挙げられるのは，Dockerfile には分岐が存在せず実行経路が 1 つしかない点である．よって網羅率という目的関数が意味をなさず，自動テスト生成の核となる探索が応用できない．したがって，Dockerfile の自動テスト生成手法には異なるアイデアが必要となる．

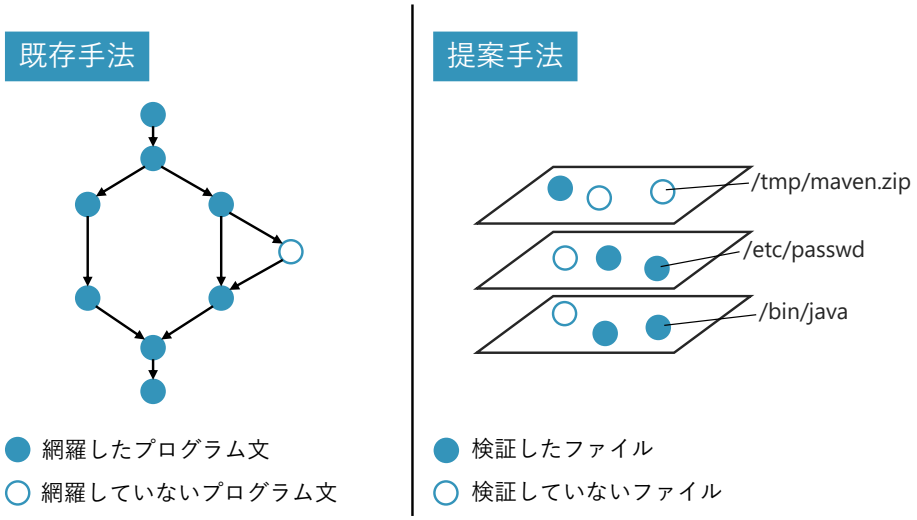


図 3: 提案する自動テスト生成手法のアイデア

3 提案手法

3.1 手法のアイデア

本研究では，Docker を利用する開発者のテスト作成支援を目的として，Dockerfile の自動テスト生成手法を提案する．提案手法の鍵となるアイデアは，処理手順ではなく処理結果に基づいてテストを生成するという点である．図 3 に既存の自動テスト生成手法と提案手法のアイデアの違いを示す．従来の Java などに対する自動テスト生成手法は，網羅率という処理手順を基準にした指標に基づいてテストを生成していた．これに対し，提案手法では Docker イメージという処理結果を基準にテストの生成を試みる．テストの品質評価においても類似したアイデアが存在する．テストの実行結果となる作用の集合を定め，その集合に対してアサーションによる検証を行った割合をテスト品質の一種だと捉えている [10][11].

2.1 節で述べたように Docker イメージの 1 レイヤは Dockerfile の 1 命令と対応しているため，Docker イメージの各レイヤの中身は Dockerfile の作用の集合だと考えられる．この全ての作用に対して期待値との一致を確認すれば，Dockerfile が期待通りに振る舞っているかを十分に確認できる．しかし，全ての作用に対してテストを生成するとテスト実行コストが高くなるだけでなく，デグレードに対して敏感なテストになってしまう．実際，自動テスト生成により生成されたテストは手動で作成したテストより脆いテストになりやすい [12]．そのため，Dockerfile 命令を用いて重要度に応じたテスト対象の選別も行う．

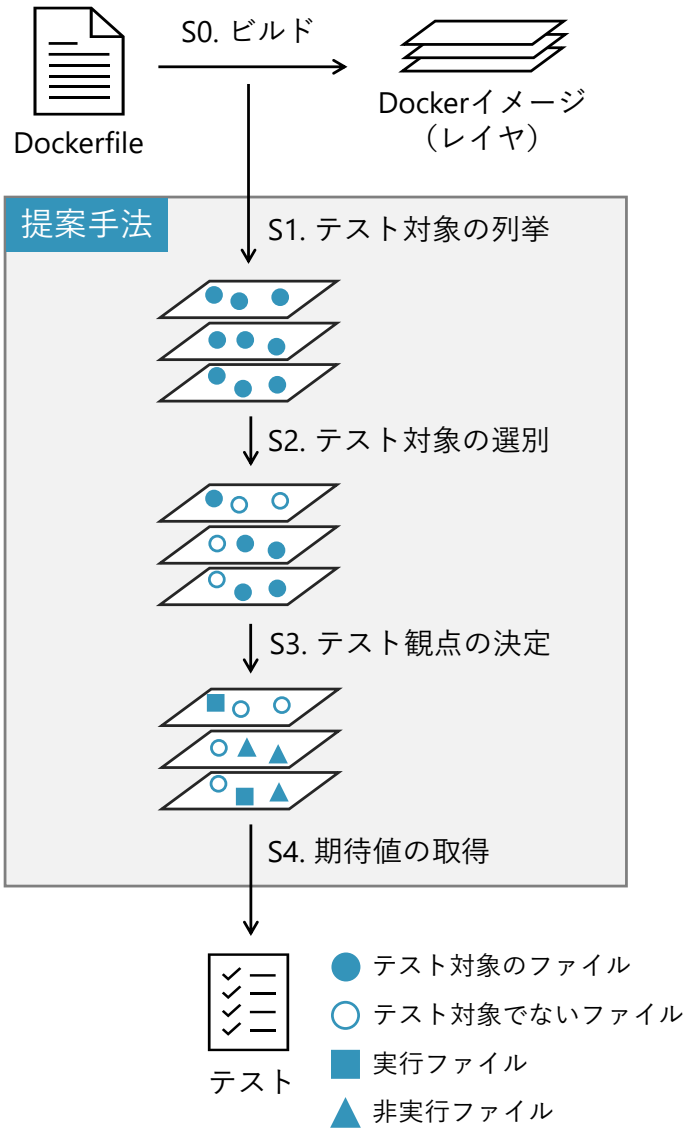


図 4: 提案手法による自動テスト生成の流れ

3.2 提案手法の流れ

提案手法による自動テスト生成の流れを 図 4 に示す．提案手法は次の 5 ステップから構成される．

S0. ビルド

本ステップでは Dockerfile から Docker イメージをビルドする．Dockerfile では “&&” を使用して 1 つの RUN 命令に複数のシェルコマンドを記述できるが，提案手法ではこれを複数の RUN 命令に分割してからビルドする．1 つのレイヤに含まれる作用を限定し，テスト対象の選別の正確さを向上させる

ためである．

S1．テスト対象の列挙

S0 で構築された Docker イメージから Dockerfile の作用を列挙する．作用には，Docker イメージ自体のメタ情報とレイヤ内のファイルの 2 種類がある．メタ情報は Docker イメージの詳細情報を表示する `docker inspect` コマンドから得られる．ファイルは Docker イメージの各レイヤから全てのファイルを列挙する．

加えて，Dockerfile 命令の解析も行う．解析する命令は，メタ情報を設定する CMD 命令や ENV 命令などの 8 命令と，レイヤを生成する FROM 命令と ADD 命令，COPY 命令，RUN 命令の 4 命令である．レイヤを生成する命令は解析後に引数などの情報を対応するレイヤに持たせる．命令の解析結果は次の S2 でテスト対象を選別する際に使用する．

S2．テスト対象の選別

S1 で列挙したテスト対象を，Dockerfile 命令の解析結果を用いて選別する．目的はテスト対象の限定によるテスト実行コスト，及びテスト全体の脆さの低減である．Dockerfile 命令の作用の数は命令の種類によって大きな差がある．例えば COPY 命令で 1 つのファイルをコピーしたレイヤのファイル数は 1 個だが，FROM 命令のレイヤや RUN 命令内でコマンドをインストールしたときのレイヤには数千ファイルが存在する．このような作用全てに対してテストを生成すると，十分だが過剰なテストになってしまう．これを解決するために，重要度に応じてテスト対象を選別する．

選別においては，ヒューリスティックに基づく得点ルールを考える．メタ情報は 2 個，ファイルは 18 個の得点ルールに基づいて加点または減点される．メタ情報の得点ルールを 表 1，ファイルの得点ルールを 表 2 に示す．得点ルールは GitHub 上で CST が用意されているリポジトリ 10 件から開発者が作成したテストを調査し，開発者が優先的にテストしていると見られる項目をルール化した．全ての作用に対する得点付けが終了した後，閾値以上の点数を持つ作用を生成の対象とする．提案手法を利用する際は，閾値の調整により生成するテストの数を調節可能である．

表 1: メタ情報に対する得点ルール

条件	得点
Dockerfile 命令で設定された	10
`docker inspect` コマンドから取得した	8

S3. テスト観点の決定

S2 で選別した作用に対するテスト観点を決定する．メタ情報は CST の metadataTest 形式で設定内容が正しいか確認する．ファイルは実行ファイルと非実行ファイルで異なるテストの検証方法を適用する．この 2 つのファイルは性質が異なり，確認すべき項目も異なるためである．実行ファイルは CST の commandTests 形式で，`which` コマンドなどによる存在確認と，オプションを使用したバージョン確認を行う．非実行ファイルは CST の fileExistenceTests 形式で絶対パスと共にファイルの存在有無を確認する．これらも S2 の得点ルールと同様に開発者の作成した CST を調査して決定した．ただし，実行ファイルは環境変数 `PATH` に設定されているディレクトリ内で実行権限があるファイル，非実行ファイルは実行ファイル以外のファイルとする．

表 2: ファイルに対する得点ルール

条件	得点
ADD 命令，COPY 命令の保存先とパスが一致する	9
ADD 命令，COPY 命令の保存先ディレクトリの配下にある	3
RUN 命令内のコマンドの引数とファイル名が一致する	5
RUN 命令内のコマンドの引数がパスに含まれる	2
ベースイメージのイメージ名やキーワードがパスに含まれる	3
コンテナ実行時の作業ディレクトリに設定されている	3
環境変数に設定されている	2
環境変数 `PATH` に設定されたディレクトリ配下にある	2
`/bin/`がパスに含まれる	3
`/etc/`がパスに含まれる	3
`/conf/`がパスに含まれる	3
ファイル名が`.sh` で終わる	3
FROM 命令から生成されている	-5
削除されている	-10
パスが`/var/lib/apt/lists/`で始まる	-10
`/tmp/`がパスに含まれる	-10
`/cache/`がパスに含まれる	-10
`/log/`がパスに含まれる	-10

S4．期待値の取得

S3 で決定したテスト観点に従って CST 形式でテストを生成するために，期待値を得る必要がある．従来の自動テスト生成と同様にボトムアップに期待値を決定する．commandTests は期待値を取得するためにコンテナに対する解析が必要となる．存在確認をする commandTests の期待値を得るには，コンテナ上で `which` コマンドなどの引数に実行ファイルのファイル名を渡して実行し，出力を確認する．実行ファイルのパスと出力が同じ場合は存在確認をする commandTests を生成し，バージョン確認をする commandTests の期待値を取得する処理に移る．実行ファイルのパスと出力された結果が異なる場合は commandTests ではなく fileExistenceTests を生成する．また，バージョン確認をする commandTests の期待値を得るには，実行ファイルのファイル名とバージョンを確認するためのオプションとしてよく使用される`--version` や`-version`，`-V` を組み合わせたコマンドをコンテナで実行する．コンテナの出力からバージョンだと推測される文字列が得られたとき，使用したオプションと得られたバージョンを用いて commandTests を生成する．ただし，metadataTest と fileExistenceTests は Dockerfile や Docker イメージの解析結果から期待値を取得できるため，コンテナに対する解析は行わずにテストを生成する．

4 実験

4.1 実験の概要

提案手法が適切にテストを生成できるか確認するために以下の 2 つの実験を行う．

実験 1：生成されたテストの十分性の調査

実験 2：生成されたテストの妥当性の評価

実験 1 では，提案手法で十分なテストが生成可能か確認するために，Dockerfile の全ての作用に対してテストが生成されているか調査する．実験 2 では，提案手法の S2. テスト対象の選別で重要度に応じたテスト対象のフィルタリグが可能か，及び生成したテストの内容が適切か確認する．

実験題材は CST を用いて Dockerfile のテストを行っている 10 個の OSS プロジェクトである．対象プロジェクトの概要を 表 3 に示す．様々な視点での CST を対象とするために各プロジェクトの主たる貢献者は全て異なるものとし，プロジェクト内に複数の Dockerfile が含まれる場合はそのうちの 1 つを実験の題材としている．また，CST により Dockerfile のテストを行っているプロジェクトに限定した理由は，実験 2 で提案手法により生成されたテストと開発者が作成したテストの比較を行うためである．開発者が作成したテスト数の合計は，metadataTest が 25 個と commandTests が 34 個，fileExistenceTests が 47 個，fileContentTests が 14 個の計 120 個である．ただし，通常 metadataTest は複数のメタ情報の検証をまとめて 1 個のテストとして数えるが，本実験では確認しているメタ情報の数を metadataTest の個数としてカウントしている．

表 3: 対象プロジェクトの概要

プロジェクト名	Docker イメージのサイズ	テスト数
zephinzer/cloudshell	29.8MB	6
corretto/corretto-docker	378MB	4
royge/deployer	413MB	8
appwrite/docker-base	1.11GB	37
jenkins-infra/docker-builder	2.11GB	36
drecom/docker-rockylinux-ruby	1.12GB	3
airdock-io/docker-sonarqube-scanner	108MB	6
googleapis/testing-infra-docker	3.8GB	9
liatrio/knowledge-share-app	125MB	5
sassoftware/viya4-iac-aws	2.32GB	6

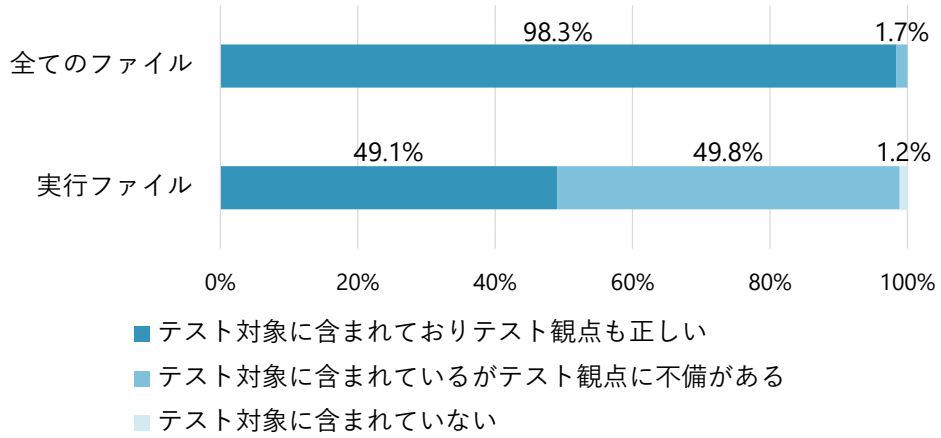


図 5: 全てのファイルと実行ファイルのうちテスト対象に含まれている割合

4.2 実験 1：生成されたテストの十分性の調査

評価方法： 本実験の目的は提案手法により十分なテストが生成可能か確認することである．そのために S2 のテスト対象の選別をしていない状態で，Docker イメージのレイヤ内のファイル全てに対するテストが生成されているか確認する．まず，提案手法により S2 が行われていない状態の Dockerfile テストを生成する．次に，全てのファイルに対してファイルの存在を確認するテストが生成されているか調査する．ただし，ファイルの存在を確認するテストとは `which` コマンドなどを使用した commandTests と，fileExistenceTests を指す．加えて，レイヤ内の実行ファイルに対しては，その実行ファイルをテスト対象とした存在確認をする commandTests と，バージョン確認をする commandTests が生成されているか調査する．

結果： レイヤ内の全てのファイルと実行ファイルのうち提案手法で生成されたテストの対象に含まれているものの割合を 図 5 に示す．まず全てのファイルの結果に着目する．テスト対象に含まれていないファイルはなかったため，ファイル全てに対して提案手法によりテストが生成されていることがわかる．次にテスト対象に含まれているがテスト観点に不備がある 1.7% のファイルに着目する．これらのファイルはテストの確認項目である存在有無の期待値が実際の存在有無と一致していないものであった．これらを確認すると，どれも Dockerfile のビルド途中とビルドが終了した時点での存在有無が異なっていた．例えば，図 6 のような Dockerfile を考える．この例はプロジェクト testing-infra-docker の Dockerfile に RUN 命令内のシェルコマンドを分割する処理を施したものから，Maven をインストールする処理を抜粋している．この処理では最初の RUN 命令で Maven を `/tmp/maven.zip` にダウンロードしているため，この RUN 命令に対応するレイヤには `/tmp/maven.zip` が存在することになっている．その後，このファイルに対する処理をいくつかしたのちに最後の RUN 命令では `/tmp/maven.zip`

```
RUN wget -q https://archive.apache.org/dist/maven/maven-3/3.8.4/
    binaries/apache-maven-3.8.4-bin.zip -O /tmp/maven.zip
RUN unzip /tmp/maven.zip -d /tmp/maven
RUN mv /tmp/maven/apache-maven-3.8.4 /usr/local/lib/maven
RUN rm /tmp/maven.zip
```

図 6: ファイルの存在がビルド途中と終了時点で異なる処理

を削除している．そのため最後の RUN 命令に対応するレイヤでは/tmp/maven.zip は存在しないことになっている．提案手法で生成されたテストではビルド終了時点での存在有無に合わせたテストを生成しているため，/tmp/maven.zip が存在しないことを確認するテストが生成されている．このため，最初の RUN 命令に対応するレイヤで存在することになっている/tmp/maven.zip は，生成されたテストの期待値と実際の存在有無が一致しなくなった．他のファイルも同等の理由であったため，ファイル全てに対してビルド終了時点での存在有無を確認するテストが生成されていることになる．

次に実行ファイルの結果を説明する．図 5 より，ほとんどの実行ファイルに対して commandTests が生成されていることが確認できる．約半数（49.1%）の実行ファイルについては存在とバージョンを確認する commandTests がどちらも生成されていた．テスト対象に含まれているがテスト観点に不備がある 49.8% の実行ファイルに着目すると，これらは全て存在を確認する commandTests は生成されているが，バージョンを確認する commandTests が生成されていない実行ファイルであった．このうち 95% はバージョンを確認するためのオプションがそもそも設けられていない実行ファイルであった．残りの実行ファイルはバージョンを確認するためのオプションを持つが，オプションや終了コードなどが提案手法で非対応の形式であったためテストの生成に失敗していた．実行ファイルのバージョン確認を--version や-version，-V のみで確認する方針には限界があると考えられる．また，commandTests のテスト対象に含まれていない実行ファイルが 1.2% 存在しているが，これらはコンテナ上で which コマンドによる所在確認ができない実行ファイルであった．具体的には，先ほど述べたようなビルド途中では存在するが終了時点では存在しない実行ファイルか，環境変数 PATH に設定されているディレクトリの順番により他の実行ファイルが優先された実行ファイルのどちらかであった．このような実行ファイルに対しては commandTests ではなく fileExistenceTests が生成されている．

4.3 実験 2：生成されたテストの妥当性の評価

評価方法： 重要度に応じてテスト対象の選別が行えているか，及び生成したテストの内容が適切か確認するために，閾値を変えながら開発者が作成した CST と同等のテストがどれだけ生成されているかを調査する．まず，提案手法を用いて閾値を変えながら Dockerfile のテストを生成する．次に，生成さ

```yaml
fileExistenceTests:
  - name: "Make"
    path: "/usr/bin/make"
    shouldExist: true
    isExecutableBy: "any"
```

(a) fileExistenceTests で存在確認をするテスト

```yaml
commandTests:
  - name: 'check make'
    command: 'which'
    args: ['make']
    expectedOutput: ['/usr/bin/make']
```

(b) commandTests で存在確認をするテスト

図 7: 同等のテストとするテストの例

れたテストに開発者が作成したテストと同等なテストがあるか目視で確認し，テスト生成数や適合率・再現率を計測する．本稿では，テスト生成数についてはプロジェクト docker-base に対する結果のみ示す．このプロジェクトの Docker イメージのサイズが実験題材の中での平均値に近いため，一例として提示する．また本実験では，内容が完全一致するテストだけでなく，図 7 のようにテストの意図が同じであれば同等のテストだと判断している．なお，本実験では開発者が作成したテストを正解データとみなしているが，開発者のテストが必ずしも良いテストとは限らない点に注意されたい．

結果： プロジェクト docker-base に対するテスト生成数を 図 8 に示す．対象の Dockerfile はイメージサイズが 1.11GB，ファイル数が 14,735 個の Docker イメージを構築する．閾値の調節によりテスト生成数を約 1 万個から数個程度まで調節可能なことが確認できる．ファイルを対象としたテストは選別の影響を大きく受けるためこれらのテストに着目すると，閾値を 0 から 10 まで動かすと生成数は 9,865 個から 16 個まで減少し，閾値が 11 以降では 0 個となっていた．他の Dockerfile に対しても同様にテスト生成数を十数個や数個程度まで選別可能であることを確認した．

次に，提案手法で生成したテストの開発者が作成したテストに対する適合率と再現率を 図 9 に示す．これは実験題材である 10 個のプロジェクト全体の結果である．再現率のグラフより，閾値が 0 のとき開発者が作成したテストの 8 割以上を提案手法でカバー可能であることがわかる．また閾値を 10 まで上げた場合でも再現率が 0.45 となっており，開発者が作成したテストのおよそ半分と同等なテストを生成していた．

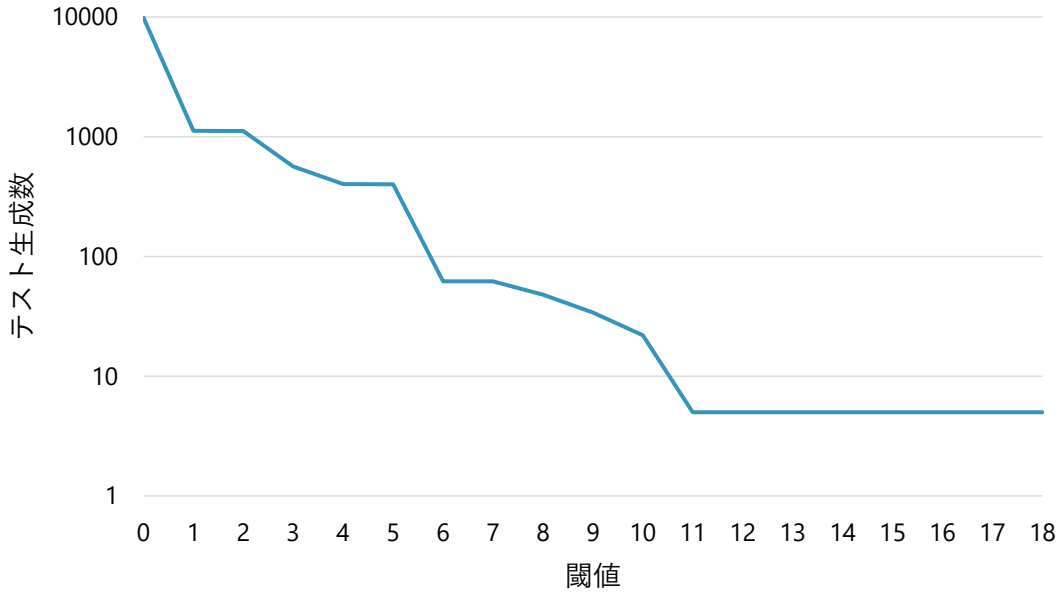


図 8: プロジェクト docker-base に対するテスト生成数

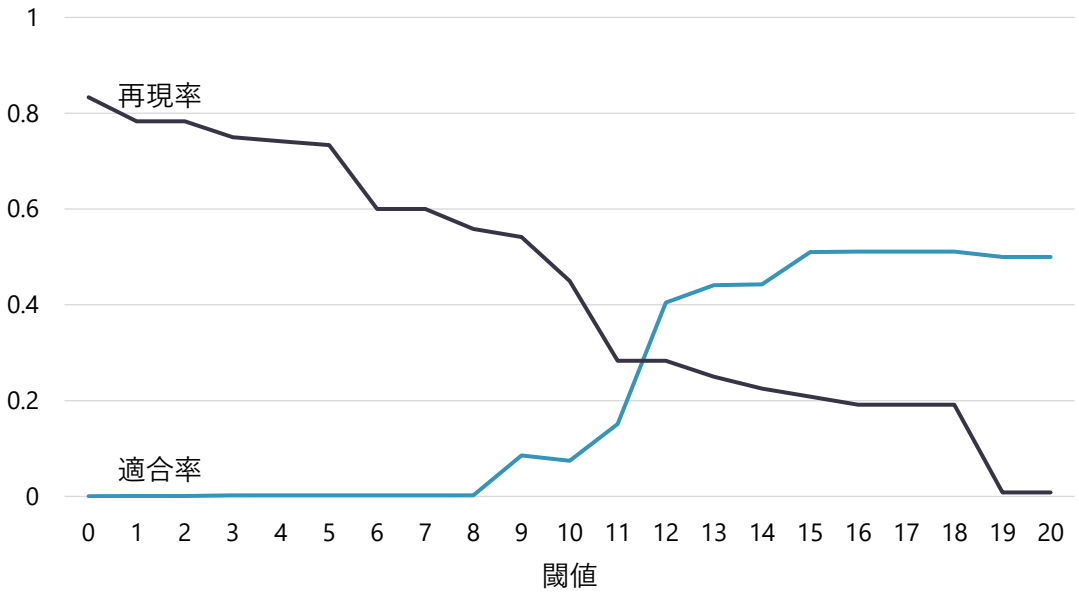


図 9: 開発者が作成したテストに対する適合率と再現率

```yaml
commandTests:
  - name: "jq installation"
    command: "jq"
    args: ["--version"]
    expectedOutput: ["jq-1.6"]
```

(a) 開発者が作成したテスト

```yaml
commandTests:
  - name: 'check jq version: RUN mv jq-linux64 /usr/local/bin/
      jq'
    command: 'jq'
    args: ['--version']
    expectedOutput: ['1\.6']
```

(b) 提案手法が生成したテスト

図 10: 開発者が作成したテストと同等のテストを生成できた例

提案手法により開発者が作成したテストと同等のテストを生成できた例を 図 10 に示す．開発者が作成したテストである 図 10(a) と提案手法が生成したテストである 図 10(b) を見ると，どちらも `jq` コマンドのバージョンを確認するテストである．よって，提案手法により開発者が作成したテストと同等のテストを生成できたと言える．

一方，開発者が作成したテストのうち提案手法で生成できなかったテストは 19 個あった．この 19 個のテストを分析した結果，以下の 3 つに分類できた．

1. ファイルの記述内容を検証するテスト：14 個
2. 存在やバージョン確認以外の commandTests：4 個
3. 結合テストのような commandTests：1 個

開発者が作成したテストのうち同等のテストを生成できなかったテストの例を 図 11 に示す．提案手法ではテスト観点をファイルの存在確認やバージョン確認のみに限定しているため，図 11(a) のようなファイルの中身を確認するテストや，図 11(b) のようなバージョン出力以外の処理を確認するテストは生成できない．よって設定ファイルの内容検証や，シェルコマンドの詳細設定の確認などを行うことは現時点ではできない．本稿ではこれらの検証を対象外としたが，テストの十分性を確保するためには必須課題であるといえる．加えて，図 11(c) に示す環境変数とバージョンの確認を同時に行う結合テストのようなテストも生成できなかった．提案手法では単体テストの生成をしているため，環境変数の設定を確認する metadataTest と `java` コマンドのバージョンを確認する commandTests は生成できるが，

```yaml
fileContentTests:
  - name: "user jenkins exists"
    path: "/etc/passwd"
    expectedContents: [".*jenkins:x:1000:1000.*"]
```

(a) ファイルの記述内容を検証するテスト

```yaml
commandTests:
  - name: 'PHP intl'
    command: "php"
    args: ["-r", 'print(\Normalizer::FORM_D);']
    expectedOutput:
      - "4"
```

(b) 存在やバージョン確認以外の commandTests

```yaml
commandTests:
  - name: "JAVA_HOME points to the correct directory."
    command: "$JAVA_HOME/bin/java"
    args: ["-version"]
    expectedError: ["Corretto-23.*"]
```

(c) 結合テストのような commandTests

図 11: 開発者が作成したテストのうち生成できなかったテストの例

これらを組み合わせたテストは生成できない．

また，図 9 の適合率は最大でも 0.5 程度であったが，これは開発者が作成したテストが十分でないことが原因だと考えられる．開発者が作成したテストでは確認できていない作用のテストも提案手法では得点が高くなっていた．例えば，開発者が作成したテストでは `apt-get install` でインストールしたコマンドでも存在を確認していない場合があったが，提案手法ではそれらのコマンドの存在を確認するテストも生成されていた．加えて，開発者は commandTests は `which` コマンドなどを用いた存在確認をするテストか，バージョン確認をするテストのどちらか一方のみを記述する場合がほとんどだが，提案手法ではその両方を生成している．存在とバージョンの両方を確認することにより，テスト失敗時の原因がコマンドの不在によるものか，バージョンの違いによるものかの判断が容易になる．したがって，提案手法で生成されたテストを用いると問題の特定と解決が効率的に行えると考える．

5　妥当性への脅威

4 節で紹介した評価実験について妥当性への脅威が存在する．本研究では CST を用いて Dockerfile のテストを行っている 10 個の OSS プロジェクトを実験題材としたが，他プロジェクトの Dockerfile を用いた場合，異なる結果が得られる可能性がある．提案手法で生成されたテストの数は Docker イメージのファイル数の影響を大きく受ける．また，開発者によってテストの対象や検証項目は様々である．したがってテストの生成数や開発者が作成したテストに対する適合率・再現率は，プロジェクトによって大きく変化する可能性がある．また，実験対象はランダムに選定し，プロジェクトのスター数や Dockerfile の行数などについて考慮していないことも妥当性に影響を与える可能性がある．他にも実際の現場で使用されている Dockerfile に対して提案手法を適用した場合，得られる結果が異なる可能性がある．

6 議論

6.1 提案手法の改善

提案手法の改善点として大きく以下の 2 つが挙げられる．1 つ目は得点ルールに改良の余地がある点である．得点ルールは，10 件のリポジトリから開発者が作成した CST を観察して決定しているが，筆者の主観であり十分とは言えない．更に多くの CST の分析により，ルールの拡充とテスト対象の選別の正確さが向上することが予測できる．

2 つ目はテスト観点の不足である．提案手法ではテスト対象の決定に着目し，ファイルに対するテストの検証項目はファイルの存在確認やバージョン確認のみに限定していた．しかし，Dockerfile 命令の処理内容によっては，ファイルの記述内容や権限などの項目も検証すべきである．これを実現するために Dockerfile 命令，特に RUN 命令内のシェルコマンドの解析が重要となる．現時点でのシェルコマンドの解析はコマンド名と引数を取り出すだけであるが，シェルコマンドの処理内容も考慮することで処理内容に応じたテスト観点を決定できると考える．例えば，`echo` コマンドと `tee` コマンドを使用してファイルの内容を変更していると把握できた場合，そのファイルは fileContentTests を使用してファイルの内容が正しく変更されたか検証すべきである．このようにしてテスト観点を充実させることで，現時点では検証できない設定ファイルの内容なども提案手法で生成されたテストを用いて検証可能になると考える．

6.2 生成されたテストの有効性の調査

本稿では 4.3 節で開発者が作成したテストを正解データとみなして実験を行ったが，これらは必ずしも良いテストとは限らない．4.3 節の実験では提案手法で生成したテストの妥当性を評価することはできたが，実際に Dockerfile のデグレードを検知可能かという有効性は評価できていない．したがって生成されたテストの有効性を調査するために，実践的な状況での評価も必要である．具体的には，まず GitHub 上から Dockerfile にデグレードが発生したコミットを調査する．次にそのデグレードが発生する前の Dockerfile に対して提案手法を用いてテストを生成する．最後にデグレードが発生した後の Dockerfile に対して先ほど生成したテストを実施し，デグレードを検知できるか確認する．これにより提案手法が生成するテストの有効性を調査できる．

7　関連研究

我々の研究グループでは，本研究と同様に Docker イメージのレイヤ構造に着目した Dockerfile の自動テスト生成手法を提案している [13]．本研究との差は大きく以下の 2 点である．1 点目は，テスト対象の選別方法である．先行研究の手法ではテスト対象を選別する際に各レイヤ内のファイルを優先度に応じて 3 つのリストに振り分ける．この 3 つのリストで要素を持つリストのうち，最も優先度の高いリストに含まれる要素を全てテスト対象としている．したがってレイヤ内での優先度しか見ておらず，Docker イメージ全体での順位付けは行えていない．これに対して本研究の提案手法では全ての作用に対して得点付けを行い，閾値によるテスト生成数の調節を可能にしている．これにより Docker イメージ全体での重要度を考慮できるほか，提案手法を使用する開発者は好みに応じてテスト生成数を調節可能となっている．

2 点目は，テスト観点の違いである．先行研究の手法ではテスト観点を `which` コマンドを使用した存在確認をする commandTests と，fileExistenceTests のみに限定している．したがって実行ファイルのバージョンを検証できない．本研究の提案手法では先行研究の手法のテスト観点に加え，実行ファイルのバージョンの検証も行っている．実際に本研究の提案手法では 図 10(b) のようなテストを生成可能である．

8　おわりに

本研究では処理手順ではなく処理結果に基づいた Dockerfile の自動テスト生成手法を提案した．評価実験により，提案手法ではほとんどの作用に対するテストを生成可能であり，開発者が作成したテストを最大 8 割以上カバー可能であることが確認できた．

今後の取り組むべき課題として，以下の 2 つを挙げる．1 つ目は提案手法の改善である．具体的な改善案としては得点ルールの拡充が考えられる．これによりテスト対象の選別の正確さが向上すると期待される．また検証可能な項目を増加させるために，テスト観点の充実も求められる．2 つ目は提案手法により生成されたテストの有効性の調査である．本稿では生成されたテストの十分性と妥当性については評価したが，実践的な状況で活用できるかという有効性については調査できていない．そのため有効性の調査として，実際に発生したデグレードを用いた実験も行う必要があると考える．

謝辞

本研究の遂行にあたり，多くの方々にご指導とご支援を賜りました．

楠本真二教授には発表練習の場で数多くの助言をいただきました．客観的な視点からのご指摘は自身の考えを今一度整理するきっかけとなり，研究の改善に繋がりました．また研究生活の中では差し入れなど様々な場面でお気遣いいただき，日々の研究活動を快適に行うことができました．心より感謝申し上げます．

柗本真佑准教授には本研究のテーマ設定から論文執筆まで，全ての場面において多くのご指導をいただきました．特に個人ミーティングや論文添削では手厚くご指導いただき，研究の進め方だけでなく，プレゼン方法などの人への伝え方についても多くの学びを得ることができました．この 1 年で大きく成長できたのは柗本先生のおかげです．深く感謝申し上げます．

事務補佐員の橋本美砂子様には備品の管理や出張手続きなど，多岐にわたるご支援をいただきました．また普段の生活でも様々な場面でお気遣いいただきました．配属当初は不安もありましたが，橋本さんのおかげで安心して研究生活を送ることができました．ありがたいご配慮に感謝しています．

楠本研究室の先輩方には研究活動に関する助言のみならず，生活面でも多くの場面でお世話になりました．先輩方は私の手本であり，研究に取り組む姿勢などたくさんのことを学ばせていただきました．他にも普段の雑談や研究室内のイベントは良い気分転換になり，研究室では楽しい時間を過ごすことができました．心からお礼申し上げます．

楠本研究室の同期の皆様とは研究に励む仲間として切磋琢磨することができました．気軽に雑談や相談ができたことは，精神面での大きな支えになりました．研究が思うように進まず苦しいときでも踏ん張れたのは皆様のおかげです．本当にありがとうございます．

そして心置きなく研究に取り組めたのは，これまで支え続けてくれた家族のおかげです．経済面のみならず精神面でもたくさんのサポートをしていただき，感謝の念に堪えません．

最後に，本研究を支えてくださった全ての方々に改めて感謝申し上げます．

参考文献

[1] Sharma, P., Chaufournier, L., Shenoy, P. and Tay, Y. C.: Containers and virtual machines at scale: A comparative study, in *Proceedings of International Middleware Conference*, pp. 1–13 (2016).

[2] Wu, Y., Zhang, Y., Xu, K., Wang, T. and Wang, H.: Understanding and predicting Docker build duration: An empirical study of containerized workflow of OSS projects, in *Proceedings of International Conference on Automated Software Engineering*, pp. 1–13 (2023).

[3] Boettiger, C.: An introduction to Docker for reproducible research, *Journal on Operating Systems Review*, Vol. 49, No. 1, pp. 71–79 (2015).

[4] Henkel, J., Silva, D., Teixeira, L., d'Amorim, M. and Reps, T.: Shipwright: A human-in-the-loop system for Dockerfile repair, in *Proceedings of International Conference on Software Engineering*, pp. 1148–1160 (2021).

[5] Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., Harman, M., Harrold, M. J., McMinn, P., Bertolino, A., et al.: An orchestrated survey of methodologies for automated software test case generation, *Journal of systems and software*, Vol. 86, No. 8, pp. 1978–2001 (2013).

[6] McMinn, P.: Search-based software test data generation: a survey, *Journal on Software testing, Verification and reliability*, Vol. 14, No. 2, pp. 105–156 (2004).

[7] Fraser, G. and Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software, in *Proceedings of Symposium and European Conference on Foundations of Software Engineering*, pp. 416–419 (2011).

[8] Fraser, G. and Arcuri, A.: A large-scale evaluation of automated unit test generation using EvoSuite, *Transactions on Software Engineering and Methodology*, Vol. 24, No. 2, pp. 1–42 (2014).

[9] Lukasczyk, S. and Fraser, G.: Pynguin: automated unit test generation for Python, in *Proceedings of International Conference on Software Engineering*, pp. 168–172 (2022).

[10] Koster, K. and Kao, D. C.: State coverage: A structural test adequacy criterion for behavior checking, in *Proceedings of Joint Meeting of the European Software Engineering Conference and Symposium on The Foundations of Software Engineering*, pp. 541–544 (2007).

[11] Schuler, D. and Zeller, A.: Checked coverage: an indicator for oracle quality, *Journal on Software Testing, Verification and Reliability*, Vol. 23, No. 7, pp. 531–551 (2013).

[12] Fewster, M. and Graham, D.: *Software Test automation: Effective use of test execution tools*, Addision-Wesley (1999).

[13] 岩瀬匠：OverlayFS の解析による Dockerfile のテスト自動生成 (2024).